09/893385

End of Result Set

Generate Collection Print

L17: Entry 2 of 2 File: TDBD Nov 1, 1990

TDB-ACC-NO: NB901154

DISCLOSURE TITLE: Global Insertion and Iterative Refinement of Breakpoints in Logic Synthesis Verification.

PUBLICATION-DATA:
IBM Technical Disclosure Bulletin, November 1990, US

VOLUME NUMBER: 33
ISSUE NUMBER: 6B
PAGE NUMBER: 54 - 57

PUBLICATION-DATE: November 1, 1990 (19901101)

CROSS REFERENCE: 0018-8689-33-6B-54

DISCLOSURE TEXT:

- Logic synthesis may be used to translate a netlist representation of a logic design into an alternative netlist representation, e.g., from technology-independent to technology-dependent. One way to verify the correctness of this translation is to compare the original netlist ('model 1') with the new netlist ('model 2') to ensure that they are logically equivalent. A compare program may be written to perform this boolean comparison. - The compare program may operate by decomposing each model into segments of combinational logic. A segment consists of a single net - either a latch output or a primary output - and all the logic that feeds it, stopping at other latch outputs or primary inputs. Provided that nets have signal names such that all points across both models having the same name are logically equivalent, the compare program may associate each model 1 segment with its corresponding model 2 segment by matching signal names. The program may similarly match each input of the model 1 segment with each input of the model 2 segment. This process is called segmentation. - Considering each pair of matched segments in turn, the program may compare the two segments by simulating logic patterns on their inputs and verifying that each gives the same output value in every case. Verification is complete when all segment pairs have been considered. - A segment in a typical large design may have many hundreds of inputs and contain several thousand gates. The processing time required to simulate all possible logic patterns on the inputs to this segment will typically exceed 5 minutes. The design need have only 100 such segments for the total processing time to become unreasonable. This is referred to as the 'large segment problem'. - One approach to the large segment problem is to insert 'breakpoints' in each model prior to segmentation. The compare program may treat breakpoints as it does primary inputs, primary outputs and latch outputs, and use them in defining the segments. Thus, a large segment may be broken down into two smaller segments by inserting a breakpoint somewhere within it. Processing time is approximately related to the square of the number of inputs to the segment, so if the inputs can be halved by adding a breakpoint, then the total processing time is halved (there are now two segments to consider). - Breakpoints may be inserted manually by examination of the model 1 and 2 netlists. Candidate points are nets that have the same signal name in both models. Manual insertion is labor-intensive, requires expert knowledge of the design and typically does not provide an effective set of breakpoints. A program may be written to analyze the structure of the design to determine suitable breakpoints, but this typically requires a complex algorithm and long processing time, and again is not always fully effective. - An effect of adding breakpoints at points of signal name match between the two models is that certain choices of breakpoint may lead to segments falsely appearing to be non-equivalent. Such 'miscompares' typically occur when breakpoints are inserted in logic that has been altered during synthesis by factorization, application of de Morgan's law or removal of redundancy. In these situations, a breakpoint causes each segment of a pair to have a different set of inputs. The compare program treats input as independent variables and so finds the segment pair non-equivalent. - With the disclosed technique, breakpoint insertion problems become trivial since breakpoints are inserted at every point of signal name match between two segmented representations of the logic design. This provides maximum reduction in segment sizes and facilitates comparison of the representations for logical equivalence. Breakpoints are not inserted on blocks that have fewer than two inputs, have no outputs or that already define segment boundaries. An iterative process of breakpoint refinement is also described which eliminates those breakpoints that cause segment pairs to falsely appear to be non-equivalent. These techniques have the following advantages over existing methods: they allow complete comparison coverage to be achieved, processing time is reduced, no expert knowledge of the particular design is required and the algorithms used are simple. - The use of breakpoint insertion to tackle the large segment problem requires a trade-off between breakpoint effectiveness and the risk of causing miscompares. Both of these factors increase with the number of breakpoints added. For a large design it is likely that a quantity of breakpoints sufficient to alleviate the large segment problem will cause miscompares. Therefore, one requirement of breakpoint insertion is that a technique is available to determine which breakpoints are responsible when miscompares occur and to eliminate such breakpoints. Given such a technique, it is logical to insert as many breakpoints as possible. In fact, maximum effectiveness of breakpoint insertion may be guaranteed by inserting a breakpoint at every point of signal name match between the two models; this is referred to as global breakpoint insertion. A program may be created to read the netlists and generate a list of candidate points for each model. To avoid unnecessary effort the program should ignore blocks that have fewer than two inputs (this prevents every inverter in a chain being given a breakpoint), or have no outputs, or already define segment boundaries, i.e., latches, drivers, receivers, transceivers. - The two lists of candidate points are matched by signal name to generate a list of 'global breakpoints' which are added to the models before running the compare program. - Breakpoint refinement is required when miscompares appear after breakpoints are inserted. Analysis of the results of the compare program allows a determination of which breakpoints are causing the miscompares. These 'bad' breakpoints can then be deleted from the models and the compare program rerun. This process is repeated until either no miscompares remain, in which case comparison is complete, or some remain but it can be shown that they are not caused by any breakpoints. In the latter case the miscompares are genuine non-equivalences. This process is known as iterative breakpoint refinement. - Breakpoints, along with latch outputs and primary inputs and outputs, define the boundaries of segments. It follows that, if a segment pair miscompares because of a breakpoint, that breakpoint must be an input to at least one segment of the pair. The first step in determining bad breakpoints is to extract from the results file the inputs of all miscomparing segment pairs and to find which of these inputs are breakpoints. Of those so found, some might have been "don't care" in the logic patterns that caused the miscompares to show. This happens when a breakpoint is an input to a gate controlled by another input. "Don't care" breakpoints can be ignored since they could not have caused the miscompares and the following assumes that they are ignored. Any breakpoints that are an input to only one segment of a pair will have been marked specially. These are unconditionally bad - a segment pair with different inputs to each segment, one of these inputs a breakpoint, can never be equivalent - and are prime candidates for deletion. - If no such marked breakpoints are found in the inputs of the miscomparing segments then all the breakpoints that are found become candidates for deletion. It may so happen that a segment pair contains two breakpoints, only one of which is bad. It is important to avoid deleting both since removal of good breakpoints defeats the objective of reducing segment sizes. One measure of the likelihood of a breakpoint being bad is the number of miscomparing segment pairs in which it appears. A breakpoint appearing in many segment pairs should be deleted in preference to one that appears a few times. - Determination of bad breakpoints when no marked ones are found is facilitated by drawing a matrix of miscomparing segment pairs versus breakpoints found in their inputs. The diagram shows an example of such a matrix which enables visual detection of patterns in the breakpoints. Good candidates for deletion are those that appear many times compared to others, or those in a group whose members encompass all the miscomparing segment pairs. Breakpoints that remain after iterative breakpoint refinement will be an optimal set providing maximum reduction of segment sizes. - BREAKPOINT MATRIX DIAGRAM Breakpoints Model 1 Model 2 AAABUSY10 HOG24BR10 .AAABUTC10 HOGUBBC10 . AAABVTV10 HOG34CQ10 . .AAABVUH10 HOG34CG10 . . AAABVUT10 HOGOOBC10 . . .AAABVVR10 HOG36CL10 . . . AAABVWD10 HOG36CG10 . . . .AAABVWV10 HOG36BW10 . . . . AAABVXH10 HOGVIAB10 . . . . .AAABVXT10 HOG38BH10 . . . . . AAABVYF10 HOG38BM10 . . . . . .AAABVYX10 HOG3NAL10 . . . . . . AAABVZD10 HOGZZBW10 Segment pairs . . . . . . .AAABVZJ10 HOGZZBR10 Model 1 Model 2 . . . . AAABWCU10 HOGZ8BH10 . . . . .AAABWDC10 HOGZ7AG10 AAACGBB10 HOGX6CG10 X XXX AAACGBD10 HOGX6DM10 X XXX AAACGBL10 HOGOGBM10 X X AAACGBR10 HOGVIBM10 X X AAACGBX10 HOGOJBR10 X X AAACGBZ10 HOGOJDH10 X X AAACGDI10 HOGOIBC10 X X X AAACFDK10 HOGOHBC10 X X X AAACGDM10 HOGWJDM10 X X AAACGDQ10

HOGZZAG10 X XXX AAACGIN10 HOGOABH10 X XXX X AAACGIT10 HOGPTDR10 X X AAACGIV10 HOGOCDC10 X X AAACGKE10 HOGOBBC10 X X AAACGKG10 HOGYXAQ10 X XX XX An X shows where a breakpoint (identified by its model 1 and model 2 names) is an input to a segment pair. Breakpoints AAABUSY10 HOG24BR10 and AAABUTC10 HOGUBBC10 between them encompass all the segment pairs and would be good candidates for deletion.

WEST Search History

```
DATE:  Thursday, November 13, 2003
```

Set Name side by side	Query	Hit Count	Set Name result set
	DB=USPT,PGPB,JPAB,EPAB,DWPI,TDBD; PLUR=YES; OP=ADJ		
L19	L8 and (bad break?point)	0	L19
L18	L8 and (bad breakpoint)	1	L18
L17	L8 and (useful breakpoint)	3	L17
L16	L8 and (useful break?point)	0	L16
	DB=USPT; PLUR=YES; OP=ADJ		
L15	L14 and (remove$ near2 (unwant$ or bad or unuse$))	1	L15
L14	L12	39	L14
	DB=USPT,PGPB; PLUR=YES; OP=ADJ		
L13	L11 and bad near2 (break?point or breakpoint)	1	L13
L12	L9 and l6	71	L12
L11	L8 and l6	105	L11
	DB=USPT,PGPB,JPAB,EPAB,DWPI,TDBD; PLUR=YES; OP=ADJ		
L10	(break?point or breakpoint) near2 (refine$)	9	L10
L9	(break?point or breakpoint) near2 (remov$ or delete$)	170	L9
L8	(instrument$ or break?point or breakpoint or checkpoint or check?point) near2 (remov$ or delete$)	10024	L8
	DB=USPT,PGPB; PLUR=YES; OP=ADJ		
L7	L5 or L4 or L3 or L2	11089	L7
L6	L5 or L4 or L3 or L2 or L1	12677	L6
L5	(((714/34 |714/35 |714/36 |714/37 |714/38 |714/39 |714/40 |714/41)!.CCLS.))	1852	L5
L4	(((712/227 |712/228 |712/229)!.CCLS.))	968	L4
L3	(((709/223 |709/224 |709/225 |709/226 |709/227 |709/228)!.CCLS.))	8000	L3
L2	(((702/118 |702/119 |702/122)!.CCLS.))	424	L2
L1	(((717/124 |717/125 |717/126 |717/127 |717/128 |717/129 |717/130 |717/131 |717/132 |717/133 |717/141 |717/142 |717/146 |717/154 |717/155 |717/156 |717/157 |717/158)!.CCLS.))	1909	L1

```
END OF SEARCH HISTORY
```